NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DATA ACQUISITION VIA RS-232 AND UNIVERSAL SERIAL BUS FROM A FIELD PROGRAMMABLE GATE ARRAY

by

Xinyang Liu

December 2009

Thesis Advisor:	Alexander L. Julian
Second Reader:	Roberto Cristi

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Data Acquisition via RS-232 and Univeral Serial Bus From a Field Programmable Gate Array		**5. FUNDING NUMBERS**	
6. AUTHOR(S)			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

The Field Programmable Gate Array (FPGA) offers a flexible solution for transferring data obtained from hardware to a PC for analysis and storage. Traditionally, an RS-232 serial interface is used to connect hardware to the PC. The RS-232 protocol is straightforward to implement with minimum hardware support. However, the presence of the Universal Serial Bus (USB) protocol has largely replaced traditional RS-232 communications, mainly due to a higher data rate and ease of configuration. In this thesis, models for the FPGA to send data in memory via RS-232 and USB were developed, as well as programs on the PC to accept those data streams. By using timing alone (no bidirectional communication), the RS-232 model could send data up to 36.9 kbps where as the USB model achieved a data rate of 702.6 kbps. By using the TXE pin and writing data only when the chip was ready, a 3.17 Mbps transfer rate was achieved.

14. SUBJECT TERMS RS-232, USB, FPGA			**15. NUMBER OF PAGES** 73
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DATA ACQUISITION VIA RS-232 AND UNIVERSAL SERIAL BUS FROM A FIELD PROGRAMMABLE GATE ARRAY

Xinyang Liu
Lieutenant, United States Navy
B.S., Rensselaer Polytechnic Institute, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2009

Author: Xinyang Liu

Approved by: Professor Alexander L. Julian
Thesis Advisor

Professor Roberto Cristi
Second Reader

Professor Jeffrey B. Knorr
Chairman, Electrical and Computer Engineering Department

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A Field Programmable Gate Array (FPGA) offers a flexible solution for transferring data obtained from hardware to a PC for analysis and storage. Traditionally, an RS-232 serial interface is used to connect hardware to the PC. The RS-232 protocol is straightforward to implement with minimum hardware support. However, the presence of the Universal Serial Bus (USB) protocol has largely replaced traditional RS-232 communications, mainly due to a higher data rate and ease of configuration. In this thesis, models for the FPGA to send data in memory via RS-232 and USB were developed, as well as programs on the PC to accept those data streams. By using timing alone (no bidirectional communication), the RS-232 model could send data up to 36.9 kbps where as the USB model achieved a data rate of 702.6 kbps. By using the TXE pin and writing data only when the chip was ready, a 3.17 Mbps transfer rate was achieved.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

BRAM	Block Random Access Memory
DCE	Data Circuit Equipment
DTE	Data Terminal Equipment
FIFO	First In First Out
FTDI	Future Technology Devices International
FPGA	Field Programmable Gate Array
PC	Personal Computer
USB	Universal Serial Bus
VCP	Virtual Communications Port
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

EXECUTIVE SUMMARY

With the amount of data that today's sensors can gather, data needs to be quickly offloaded from the sensor to a storage device for analysis and storage. Initial benchmarks consisted of a data stream running at approximately 32 kbps, and while it may seem slow in terms of a gigabit Ethernet, it is not trivial in terms of embedded applications. Furthermore, the timing and synchronization are not done via a simple command in software, but must be programmed with logical components. An FPGA was used to tie the external sensor boards with the PC for rapid development and flexible configuration.

Two approaches were compared to transfer the data from the FPGA to the PC. A traditional RS-232 serial exchange was first used for its simple protocol and its availability. It was found that the top speed using this protocol achieved 36.9 kbps, slightly above the required data rate. Data rates faster than 36.9 caused errors to be transmitted, regardless of extra added idle time in the end of each word for padding and synchronization.

The second approach was to connect the FPGA with the PC using an USB interface. This approach yielded a data rate of 702.6 kbps using a fixed write period. Attempts to increase the data rate yielded errors as well, although not as many. Synchronization was again the issue with higher data rates, due to pauses required to transfer data in the USB buffer. However, by monitoring the TXE pin and writing data only when the chip was ready, a 3.1 Mbps transfer rate was achieved.

This thesis demonstrated that data transfer is possible from a FPGA to a PC for data rates not exceeding what was achieved. The results are generally applicable to any devices that want to offload data to a PC. It improved on exsiting solutions by offering a easy to use and maintain development environment without risking proprietary vendor locking. By using more stringent flow control in the future, the transfer rate could be improved to provident services to a wider range of applications.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I’d like to thank my advisor, Professor Julian, for guiding me through this thesis and putting difficult problems in plain concepts. Thanks to my family, for their interest, and friends, for their encouragement.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. RESEARCH GOAL

The goal of this research was to develop a data acquisition solution to transfer data from a Field Programmable Gate Array (FPGA) to a Personal Computer (PC). Data from sensors are often analyzed in real time with an oscilloscope but rarely stored in raw digital form. Depending on the application, a large amount of data may need to be moved very quickly from the hardware to PC memory. This research is aimed at developing an efficient way to transfer that data on the PC, where it can be stored and all the tools of analysis can be put to use.

The programs for both RS-232 and USB had to be developed because there are no premade implementations available from the venders. There was a sample program packaged with the USB drivers for Linux. That sample program read and wrote to the USB FIFO chip from the PC. However, no mechanism exists to get meaningful data to the FIFO.

B. MOTIVATION

The original motivation for this thesis was to take data from digital sensors and offload it to the PC. The initial data stream was two channels with 10,000 16-bit data points per channel every 10 seconds. A mechanism was needed to transfer data at 32+ kbps to the PC for calculations and storage. This requirement was later dropped to make the system applicable to as many devices as possible.

C. BACKGROUND

The RS-232 standard specifies all aspects of communications from connector types to voltage levels, and signals used. It is an extensive standard, and pertinent information will be presented here.

The physical connection is between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE). In this case, the DTE is the PC and the DCE is the FPGA. The pins on the serial cable are labeled 1 through 9, with the important pins being 2 and 5. Pin 5 is the ground pin that the voltages of the other pins are compared against. Pin 2 is the Receive Data (RxD) pin, which receives data from the FGPA to the PC. Pin 3 could be used later should a need arise to send data to the FPGA from the PC. There are also handshaking signals on other pins that signals data is ready to be sent and the equipment is ready to receive [1]. However, those mechanisms are not used in the current design because the link is assumed to be ready, as triggered by running the program itself on the PC side and an explicit trigger on the FPGA side.

The signal operates between +/- 5V, with +5 being 0 and -5 being 1. When the line is idle, a 1 is asserted on RxD. When the transmission starts, the Start Bit of 0 is asserted. Then a set of data bits are transmitted, with the least significant bit first. When the data bits have finished transmitting, a Stop Bit of 1 is asserted, effectively idling the line. When the next set of data is ready for transmission, the Start Bit is asserted again and the pattern repeats [1]. The number of bits used for the data and the stop bits can vary. For this thesis, the number of data bits was set to 8 and the number of stop bits was set to 1. Later, the stop bits were padded so the period of each data word is a power of 2, for a total period of 16 bits.

To connect to the PC, a USB to RS-232 converter was used. The converter changed the physical connection of a female DB-9 connector to a Type A USB plug. The driver on the PC then converted the USB connection into a Virtual Communications Port, which made the USB connect look like a serial port for all other applications on the PC. This choice was made because RS-232 connectors are being phased out by PC manufactures and it was desired that the data be comparable regardless of future equipment. The adaptor effectively only changed the physical connection of the device, with all other aspects of the setup working exactly as if the RS-232 cable was directly connected to the serial port on the PC.

The USB standard first came out in 1996 with USB 2.0 coming out in 2000 [2]. The purpose of the standard was to standardize connectors and simplify software

development. The original USB 1.0 standard could support data flow speed of 1.5 Mbps, with USB 1.1 supporting 12 Mbps and USB 2.0 supporting 480Mbps. However, due to the additional frames inserted to synchronize data flow, the maximum effective transfer rate is 320 Mbps [2]. For this thesis, the mode of operation was set to USB 1.1 by the FIFO chip, with a maximum transfer rate of 1 Mbyte per second [11]. USB devices require individual drivers to operate since each device communicates differently. These drivers were included as dynamic link libraries, or just libraries. Only the libraries were needed to make the device functional, however, to develop new programs for the device, the header file that specifies the function prototypes in the libraries were also needed. This way, the compiler will know what functions the library provides as well as the input and output data types it expects.

D. RELATED WORK

Several existing projects aim to achieve the same goal as this thesis. However, some are incomplete while others focused on the very base of the USB protocol. Other solutions consist of using the native USB port on the FPGA, but the risk of vendor lock is high. Other solutions assume the data have already loaded into the FIFO, and focus on how to program the PC to extract the data from the FIFO.

One prepackaged solution is the SRC-6 reconfigurable computer. It has two Virtex user programmable FPGAs and one controller FPGA that facilitates communications from the computer to the FPGA. External data can be connected to the FPGA and data stored on the onboard memory banks. Programming the SRC-6 could be done entirely in C, without the need of hardware description languages such as Verilog or Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). The data transfer between the onboard memory and the PC operates at a standard 800 Mbits per second with up to 1400 Mbits per second with appropriate parallelization [3]. However, the scale of the SRC-6 would be prohibitive in simple embedded applications, and its size and cost would be excessive in this case.

Various experiments were setup in the past to read data from the FPGA to the PC. One such experiment transferred images from an FPGA to the PC using an USB chip

similar to this thesis [4]. In that project, image data captured by a camera was sent to the PC via a USB transfer module similar to this thesis. In that approach, a state machine was used to control data transfer. Another similar project used 4 data acquisition boards and achieved 20Mbytes per second transfer rate [14]. However, they focused on getting the data to the FGPA and did not go into the details of transferring the data from the FPGA to the PC. These two previous works serves as excellent alternatives to this thesis, however, the solution developed here are much simpler. There is another project which also facilitates FPGA to USB transfer [5], but it is still in the beginning stages of being implemented. These projects are excellent alternatives to what is developed in this thesis. However, the programming languages used such as C++ and Python are not as integrated for numerical applications in ways Matlab and Simulink is. These projects tend to focus on the lower level of the USB protocol and require extensive knowledge in the USB protocol to maintain.

Other types of FPGAs are also available with existing USB interfaces, such as the Heron RTG001 [6], Orange Tree Zest product line [7], and the Opal Kelly XEM 3010 [8]. These projects take the other extreme and bypass all low level programming and present the user with a premade interface. However, the potential for vendor lock is high. Should a feature be needed in the future that is not within the capability of that vendor, solutions developed from a generic board will not be compatible.

Another solution is to use a USB FIFO chip as a translation device from the FPGA to the PC. Future Technology Devices International (FTDI) makes one such chip. However, there are no known designs that will produce the wave patterns required to drive the chip. This is understandable because FTDI cannot produce a solution for every possible FGPA to drive its device. However, the waveforms are straightforward in the datasheet [13]. Resources provided from FTDI focused on reading the data from the chip vice getting data into the chip. Most of the effort in this thesis is to develop a model that will give the expected signals to the FIFO so that it will relay that data to the PC.

For the PC side programming, there are examples from FTDI and third parties in various languages and platforms. The most common one was written in C/C++ [9]. The example provided in the Linux driver, written in C, servers as a model to this thesis.

However, the choice for numerical analysis of the data is Matlab and there would be additional complexity to acquire the data and analyze the data from two different programs. Furthermore, future modifications to the acquisition program would require the same development environment. Matlab presents a standard and integrated development environment compatible with the numerical nature of future applications.

Although Matlab provides support to call shared library functions, there are no available Matlab solutions to work with the FTDI chip. The translation from working C code to Matlab is a straightforward process, but there are issues that have to be overcome due to the platform difference that will be discussed in Chapter II.

Similar solutions exist for the RS-232 case. One such case uses RS-232 communications to transfer data from a digital potentiometer [15]. In that project, the work is similarly divided between a circuitry part and a PC side program side. However, the solution used a fixed circuit for a particular potentiometer, which would not be effective for data acquisition, where the board needs to be flexible to take input in a potential of different formats.

Lastly, Singh and Conrad have developed RS-232, USB, and Ethernet solutions [16]. In their work, the RS-232 solution achieved a transfer rate of 57.6 kbps and the USB solution achieved 200 kbps. Like some previous projects, this thesis also focuses on the PC side programming vice the programming on the hardware. However, all these previous efforts demonstrated the viability of FGPA to PC data transfer and provided significant insights for this thesis.

E. THESIS ORGANIZATION

The rest of this thesis will cover the systems designed to accomplish the transfer of data from a FPGA to a PC. Chapter II presents three designs on the FGPA and two programs on the PC to accomplish the task. Chapter III presents the results of the designs and the data rates achieved. Chapter IV analyzes the results and offers explanations on some unexpected parts of the results. Chapter V summarizes the results of this thesis and presents follow on topics.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DESIGN

A. OVERALL ENVIRONMENT

This thesis starts with data stored in the Block RAM (BRAM) of the FPGA. The FPGA is then programmed to off load the data to the PC, either directly via the onboard RS-232 interface, or via another USB FIFO chip to transmit via USB. The PC listens for data on its USB port, either directly in the case of the USB design or via a RS-232/USB adaptor in the RS-232 design. When the user triggers the FPGA, it begins sending data. When the data in the BRAM have been transferred, the line goes idle, waiting for the next trigger. For the test case, an array of 16384 8-bit positive integers were used.

The setup for transferring data via RS-232 is shown in Figure 1. Data from the BRAM in the FPGA is sent directly to the RS-232 received data port on pin 2. A model is developed for the FPGA to send the data in the correct sequence and time. A program is also written on the PC to receive the data and store it in PC memory correctly.

The setup for transferring data via USB is shown in Figure 2. Data from the BRAM in the FPGA is sent to the parallel-USB FIFO module via one of its expansion slots, and the FIFO module then send the data to the PC via USB. The FIFO sends the data it receives to the PC via its USB drivers to send the data to the PC. However, a model still had to be developed to send data correctly from the FPGA to the FIFO. Two methods were used. The first used purely timing, where data was sent at a fixed period. The second used feedback on the TXE pin to send data immediately once the chip is ready. Similarly a program also had to be written to receive the data from the FIFO correctly to the PC.

The FPGA used was a Xilinx Spartan-3 for development work and the model was later transferred to an Opal Kelley 3050–1500 to be integrated with a voltage sensor. The Opal Kelley board uses the Spartan-3 on the backend so only minor changes were necessary [8]. The Spartan contains all the components needed for this thesis including one megabyte of RAM, a 50 MHz clock, a RS-232 port, and expansion slots to connect to

other devices. For the test, 16384 8-bit words were used. The board has 1 Megabyte of memory in the form of 32 256-kbit memory arrays, theoretically supporting up to 1024000 8-bit words [10]. However, not all memory is available to the user and the maximum data length available was 24576 words. The FPGA also provided a push button that served as a trigger for development purposes. In an actual application, it would be replaced with either an external source or a software trigger.

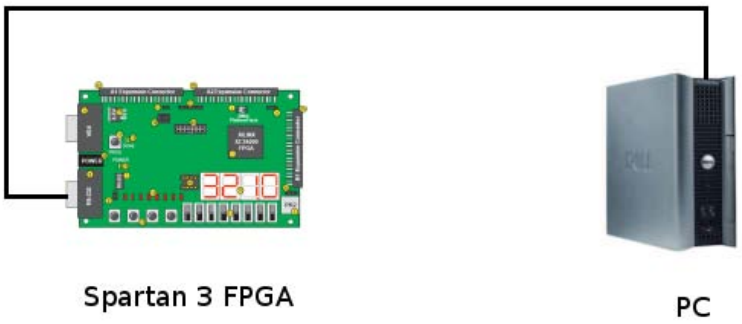


Figure 1. RS-232 data transfer diagram.

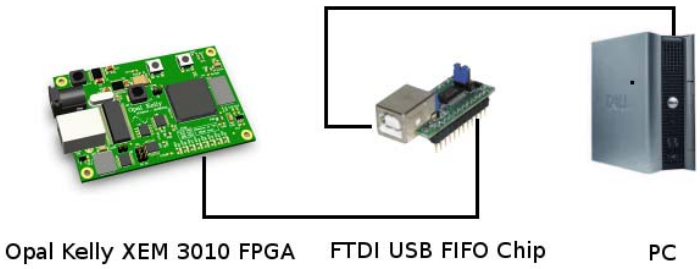


Figure 2. USB data transfer diagram.

One thing that is missing on the FPGA is a USB port, which is overcome by using the FIFO chip. The FIFO chip used was a FTDI UM245R which was initially connected to a breadboard but later permanently mounted with the FPGA. It receives data on its 8 data pins and writes to the PC via its USB port at a maximum rate of 8 Mbps [11]. The data is initially stored in the USB transfer buffer when it is initially transferred to the PC. It is then copied to the read buffer and assigned to a variable. The bus powered

configuration is used for development work to avoid extra power circuitry. However for production, self powered mode is used to isolate power requirements to the hardware itself.

For the development work, Simulink with System Generator was used to create the HDL netlist. This is then fed into ISE to be compiled into the hardware bitmap. For the PC side, Matlab with its support for shared libraries was used to read from the RS-232 port on the FPGA or the USB FIFO chip.

B. USB FPGA DESIGN USING A FIXED WRITE PERIOD

The overall design is shown in Figure 3. Once triggered, the timing counter effectively produces a 50 MHz / 2^9 = 98 KHz clock for the rest of the program. The values from the counter are used to set the WR pins and to enable the BRAM address counter. The stop trigger compares the current BRAM address value to the previous BRAM address value, and if it detects that the maximum address have been reached, it disables the timing counter, which in turn disables the address counter, and switches the data pin to idle. Details of each component are described below and the complete model is included in Appendix C.

1. Start Trigger

Without a start trigger, data will flow when the FPGA is first programmed without due regard to whether the PC is ready to receive the data, or whether the data on the FPGA is valid. A trigger mechanism is therefore needed to explicitly flag the FPGA that data is indeed ready to be transmitted. There are three way to implement the trigger, a hardware trigger, a software trigger or an external signal.

A hardware trigger is a physical piece of hardware which can be used to explicitly flag the FPGA to start transmitting. It is implemented on a push button onboard the FPGA. Because the contact time of the pushbutton is much longer than one clock cycle, an edge detector mechanism is set up to detect the falling edge of the button press. However, due to switch bouncing, there could be multiple edges for one button press. To debounce the switch, the edge detector is used to drive an 8-bit debouncing counter

initially set to 0. When the button is pressed, the debouncing counter increments, which enable the timing counter. Even if the button bounces and multiple falling edges cause multiple increments in the debouncing counter, the output is still nonzero, effectively debounces the pushbutton. Because an 8-bit counter was used as the debouncing counter, the switch could bounce up to 2^8=256 times without error.

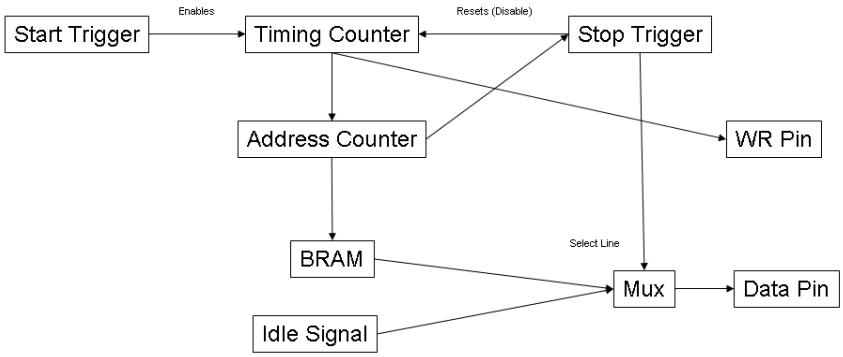


Figure 3. Block Diagram of overall USB design.

A software trigger replaces the pushbutton on the FPGA with a signal from the software interface on the PC. This has the advantage that the operator does not have to be in the same location as the FPGA, especially if the FPGA is in a dangerous location, such as where high voltage is present. This also has the added advantage of putting the entire interface in one location, on the PC.

The external trigger removes the user from triggering the flow directly. Instead of manually entering the trigger signal via a pushbutton, it listens for the signal from another device. This is the most realistic scenario in that the BRAM will be written first from another device before being transmitted to the PC. Under the previous two trigger types, the user will need to know when the data is ready to be sent. Under this type of trigger, the flow from the device to the PC will be automatic and without explicit user

intervention. However, additional handshaking will need to take place between the devices to make sure the FPGA is triggered only when it is ready.

Regardless of the type, the data flow is controlled by a start trigger and a stop trigger.

2. Timing Counter

The timing counter is effectively a clock divider. For this design, a 9-bit free running counter was used with an initial value of 128. In this setup, the counter is commanded by the clock which runs at 50 MHz on the FPGA, and competes its counting period every $2^9 = 512$ clock cycles. This effectively produces a 50 MHz / 512 = 97 KHz clock. If a specific frequency is desired, a count limited counter can be used. For example, if the frequency was desired to be 100 KHz, the count limit of 50 MHz / 100 KHz = 500 should be set.

If the value of the counter is less than or equal to 3, the WR pin on the FIFO chip is set to high. When it returns to low, it signals the FIFO that the value specified on the data pins should be read into the USB transfer buffer. The value of 3 is chose to regulate the WR pulse width. The minimal width is specified by the manufacturer to be 50 ns [11]. By choosing 3, 4 values in the timing counter are less than or greater than 3 (0, 1, 2, 3), producing a pulse width of 4/50 MHz = 80 ns.

The timing counter value is also compared to another value to enable the BRAM address counter. When the timing counter reaches this value, the BRAM address counter is enabled. This value also serves to delay the increment of the BRAM address counter until after the WR pin transitions, so the value does not change when the WR pin is active. Note that the BRAM address counter will only be enabled when the timing counter equals to the value. Therefore, it will only be enabled for one clock cycle per write cycle, and hence incrementing the BRAM address by only one per write cycle. This value could be anything after the WR pulse value (3 from the above paragraph) and the upper limit of the counter. A value of 32 was chosen so it is sufficiently different from the WR pulse. Finally, a value greater than 32 was set as the initial value of the

timing counter. This way, the counter does not set the WR pin to high due to initial conditions and increment the BRAM address counter before enabling the WR pin.

3. BRAM Address Counter

The BRAM address counter is a count limited counter with values ranging from 0 to the data length minus one. It is only incremented when the timing counter sends it the enabling signal. Because the enabling signal only last one clock period long, the BRAM address counter is only incremented once. The value of the counter is then given to the BRAM, which in turn output the value in memory. Since the value is an 8-bit integer, the value is sliced into their individual bits and sent to their respective data pins on the FIFO.

4. Stop Trigger

The stop trigger stops the data flow once the data set have been sent by comparing the current BRAM address to the previous BRAM address. If the current address is greater than or equal to the previous address, the BRAM counter is operating normally, and the WR pin is set to the output from the timing counter comparison described in section 2.B.2. However if the current address is less than the previous address, it means that the BRAM address counter have passed it maximum value and has returned to 0. In this case, the output of the WR pin is set to low, signaling that no data is to be sent. The timing counter, the debouncing counter in the start trigger, and the BRAM counter are all reset to their initial values. The timing counter is reset to set the WR pin to low and disable the BRAM counter from continuing. The debouncing counter is reset to disable the timing counter from producing WR pulses, until the start trigger is triggered again. Lastly, the BRAM counter is reset to 0, the starting point for the next data set.

C. USB DESIGN USING TXE FEEDBACK

By using the TXE pin on the FIFO, the WR pulse is only triggered when the PC is ready to receive data. In this way, the FIFO transfer at the maximum possible rate without accidental writes when the PC is not ready. The overall design is shown in Figure 4. The start and stop trigger is similar to the previous model. The TXE pin is

used to trigger a counter that will generate the WR pulse, which will in turn advance the BRAM address counter. The complete design is included in Appendix D.

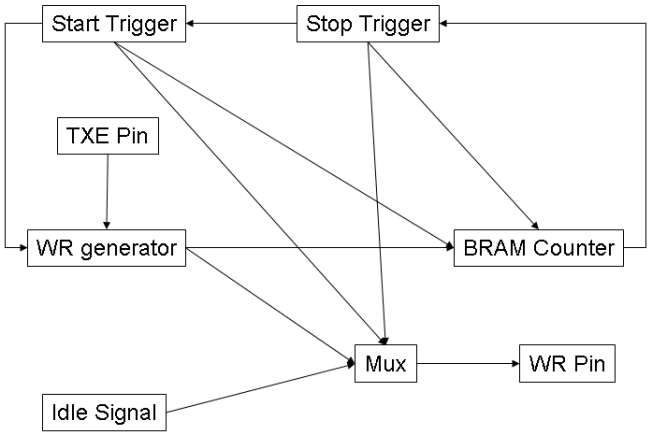


Figure 4. Block diagram of USB design using the TXE pin to regulate data flow.

1. Start Trigger

The start trigger is the same as in the previous model. However, in addition to enabling the WR generator as in the case before, its output is also used to determine the output of the select line of the multiplexer that determines the WR pin, and enable the BRAM address counter. These additional comparisons are necessary in this model in order to avoid the signal from the TXE pin continuously driving the rest of the circuit. The start trigger in this design disables that driving mechanism until the user triggers the data transfer.

2. WR Pulse Generator

The WR pulse generator produces a pulse on the WR pin when it senses that the TXE pin is low. If the TXE pin is high, hence data is not ready to be transferred, it disables the counter that generates the WR pulse and continuously resets the counter to 0,

producing an idle signal on the WR pin. When the TXE pin goes low, signaling data is ready to be written, it disables the reset and, after one clock cycle, enables a 3 bit WR generator counter. The delay is inserted so that the WR pin will not trigger while TXE is transitioning. If the WR generator counter value is between 1 and 3 inclusive, then the WR pin will go high. This allows the WR pulse to be 3 clock periods long (3*20=60 ns), similar to the previous case and within manufacture's specifications. Unlike the previous case, the BRAM address counter is enabled immediately after the falling edge of the WR pulse. This was done because the TXE pin resets the counter used to generator the WR pulse. From the results in Chapter III, the TXE goes back to high fairly quickly after the falling edge of the WR pulse. If a value much greater than 4 was used as the trigger to the BRAM address counter, the TXE pin will reset the WR counter value before that threshold value is reached, and the BRAM address counter will never advance. The concern over the BRAM address counter incrementing during write turned out to be not an issue, since the manufacture allows for the data pins to be changed immediately after the WR pulse [11].

3. BRAM Address Counter and Stop Trigger

The BRAM address counter and the stop trigger functioned similarly to the previous model. The BRAM address counter is enabled one clock cycle after the WR pulse goes low, when the WR pulse generator counter equals to 4. The index of the data is compared to its previous index and if the counter has counted to its max and returned to its initial value, the start trigger is disabled and the WR pin is set to low, signifying no additional data is present. The debounce counter is also reset, ready for the next data set.

D. USB PC PROGRAM

The program for the PC in the USB version uses libraries and header files provided by FTDI. The library provides functions such as read and write which interfaces between the hardware and user programs. The user programs then use those and other programming functions to accomplish a task, in this case, reading the data from BRAM and storing it in PC memory. The program is the same for both versions of the

FPGA model. This is because the PC program simply monitors the data buffer in PC memory and copies it to a variable. The actual transfer of data is done via the FPGA and the USB driver.

The program could be written in a wide array of languages. However, due to its numerical abilities for future applications, Matlab is chosen for this thesis. Porting from C to Matlab was rather straightforward due to Matlab's support for external libraries. However, some modifications had to be made.

The data types used in the original C program had to be recasted to allow Matlab to read purely numerical data [12]. For example, the initial header files specify the handle and data as void pointers. However, Matlab cannot deal with void pointers, so the data type had to be specified as integer pointers. Also, the data type for the actual data was initially declared as 8-bit pointers. However, Matlab treats them as strings made up of characters instead of unsigned integers. Because of this, Matlab looks for a null termination character in the end of transmission that never arrives. Therefore, the data type was redefined to be a 32-bit word and, once read into the PC, was recasted into an 8-bit quantity. In the end, the handle, along with the actual data buffer, had to be redefined to 32 bit integers, which are read correctly. These changes are included in Appendix F.

The program is divided into three parts, initialization, transfer, and closing. The complete program is included in Appendix E. During initialization, the data length is defined and the library functions are loaded. The program also defines a handle that specifies the device. The USB port connected to that device, as specified by the handle, is then opened. The existing queue is also purged of any leftover data, which is useful if there are stray bits such as when the FPGA is configured for the first time.

Once initialization is complete, the program tells the user that it is ready to receive data and asks the user to trigger the FPGA. During the transfer stage, the program continuously polls the read buffer until it is filled with the expected number of data packets. The expected number of data points specified on the PC program has to match the number of data points sent by the FPGA exactly. If the expected number is higher than the actual number sent, the PC program keeps waiting for the data

indefinitely. If the number is lower than amount sent, data is lost and the extra data corrupts the next read cycle. After the read buffer is filled, the data is assigned to a variable via the `FT_Read` function. The data variable is then typecasted back to 8-bit quantities and saved. Lastly the port is closed.

To measure the transfer time, the `tic` and `toc` commands from Matlab are used to signal the start and end of the transfer measurement. The measurement begins when the data is first detected in the read buffer, and the measurement ends after the queue reaches the expected size. However, the queue does not get filled one word at a time, but rather in packets of 4000 words at a time. Taken into account the two control words for every 62 data words, the timing actually starts after the first 3968 data words are read into the read buffer. Therefore, the time measured is not the time it took to read 16384 words, but 16384-3968 = 12416 words.

It is interesting to note that two commands are needed to read the data. The read buffer needed to be polled to make sure the correct amount of data is in the queue, and the read command is sent to read the data. When the status of the queue is read, the amount shown is the number of data words in the read buffer, which has already been transferred to the PC. The actual read command simply assigns those values to a local variable.

E. RS-232 FPGA DESIGN

The overall RS-232 design on the FPGA is shown in Figure 5. In this situation, the start trigger resets the counter to start the next write cycle, while the stop trigger serves to enable and disable the counter for the current write cycle. The counter is a 14+4+1=19-bit counter with the 4 least significant bits to control the transmission of a single word, the 5^{th} to the 18^{th} bits are use to address the BRAM, and the last bit as overflow. The stop trigger compares the current value to the maximum value of 2^{4+14}=262144, and switches the output to the transmit pin on the RS-232 port to either valid data during transmission or the idle signal when the data have finished transmitting. The details are described below with the complete schematic included in Appendix A.

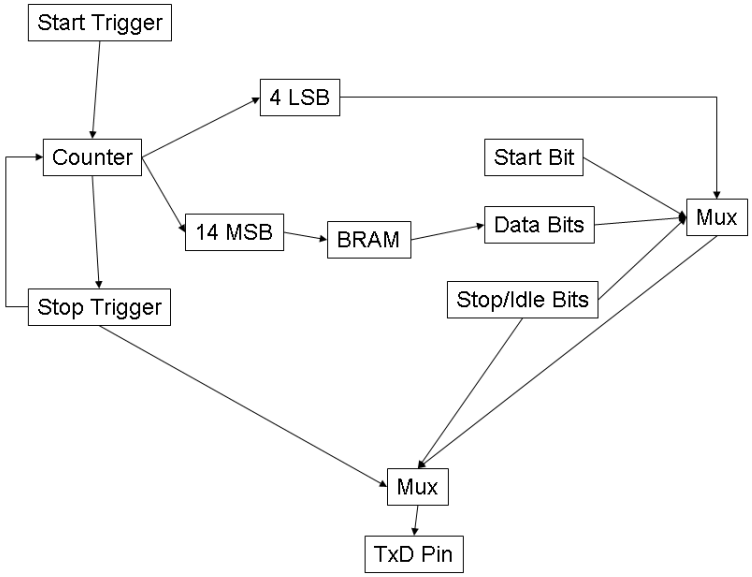


Figure 5. Block diagram for overall RS-232 design.

1. Start Trigger

The start trigger in this design is similar to the start trigger in the USB design, in that it uses a hardware pushbutton as well. In this, the trailing edge of the push button is detect and is sent directly to the reset port of the counter. This signal serves to reset the value of the counter to 0, which causes the stop trigger to enable the counter again. Note that in this model, debouncing was not implemented and turns out to be not an issue.

2. Counter

The counter used is a free running 4+14+1=19-bit counter with a period calculated from the desired baud rate. The last 4 bits are used to index individual bits to be transmitted, the middle 14 bits to index the word to be transmitted, and the MSB for overflow when the counter stops at 2^{18}. The 4 least significant bits are used as a select line to a multiplexer that have the start bit as the first data line, 8 data lines, and 7 idle lines. As the index of the counter increases, the 4 least significant bits step through the

lines of the mux and produces a proper RS-232 waveform. Seven idle bits are used so the whole sequence will fit inside 16 bits, and also was meant to allow the line a chance to resync after each word.

The next 14 bits in the counter were used to address the BRAM. Because the least significant 4 bits were used to control the transmission of each data word, the address of the data is guaranteed to remain constant for the duration of the transmission. When the least significant 4 bits overflow on to the 5^{th} bit, the BRAM address is effectively incremented. This will continue until all 14 bits are true, signifying the last data point.

After all the data are transmitted, the most significant bit is set to 1, with all other bits set to 0. This value will trigger the stop trigger, which will disable the counter. It is important to note the additional bit is necessary, because without it, the counter returns to zero and the stopping condition will never be satisfied.

The counter period is chosen to match the bit period for a given baud rate. For 38400 bps, the FPGA clock period should be scaled by a scaling factor of 50MHz/38400 = 1302. However, that scaling factor produced an incorrect bit period as measured on the oscilloscope. The scaling factor was empirically determined to be 1355.

3. Stop Trigger

The stop trigger compares the output of the counter to $2^{18}+1$. When this value is reached, all the data have been transmitted and the counter is disabled. The stop trigger also switches the output from the normal output to an explicit idle signal. In future implementations the fixed final value could be replaced with a variable value read from a PC, which is an advantage over the USB design where the final value is specified in the counter itself.

F. RS-232 PC SIDE PROGRAMMING

The program on the PC is similar to the USB version. No additional drivers are needed because Matlab have built in functions to interface with the serial driver in the

operating system. The serial port is defined, configured, opened, read, and closed. Three important configuration parameters are the baud rate, data length and timeout. The baud rate had to be defined so that the serial driver knows how long each bit pulse is. This is especially important because if the baud rate is incorrect, one bit transmitted on the FPGA could be interpreted as 2 bits on the PC. The data length is needed to be specified so Matlab will know when to stop listening on the port and write the data transferred to memory. Lastly, a timeout long enough needed to be specified to ensure the data have enough time to get transferred before the command times out. This was especially important during the diagnostic stage, where low data rates lead to transfer times that exceeded the default timeout. The completed program is provided in Appendix B.

G. SIMULATION

To simulate the expected waveforms, the start trigger connection was removed in both designs. This way, the simulation would run despite the fact that the hardware trigger could not be activated in the software simulation. In the USB design, a constant true was connected to the timing counter, and in the RS-232 case, the reset port connected to the start trigger was removed. The data was then also sent to the workspace in a structured array with time. The simulation was not done for the USB with TXE feedback because whether or not the chip is ready to receive the data cannot be determined beforehand. In other words, a button press triggering the debounce counter can be simulated in software, but a TXE waveform can not be simulated.

H. CURRENT LIMITATIONS

Currently, the data size per word is limited to 8 bits for both designs. For the RS-232 design, this is because there are only a maximum of 8 data bits per frame. For the USB case, this is because the FIFO has only 8 pins, so each transmission is limited to 8 bits. To overcome this, two memory locations could be used to store one word of data, and the values added on the PC.

Another limitation is that the data are treated as integers. This is because the data type is declared as integers in the library header file. However, using the same technique

to overcome the 8-bit data limit above, 3 bits could be used to denote the decimal point position while the other 13 bits are used to signify the value.

Also, the data are currently treated as positive quantities. Currently, when a negative value is encountered, the data is casted into zero. This limitation comes from the shared library type definition for the data that it is expecting. Since the data is declared as an unsigned quantity in the library itself, either the library have to be recompiled, taking all other dependencies into consideration, or advanced type casting will have to be implemented.

The data length currently is fixed at compilation. This means the FPGA has to be reprogrammed when the data length changes. This could be fixed in concept by allocating another set of BRAM for run time parameters. The value of the data length could be read during run time and fed into the comparator to determine whether all of the data have finished transferring.

I. CHAPTER SUMMARY

This chapter explained the design parameters for the Simulink model and the Matlab programs. The next chapter will provide results obtained with these designs.

III. RESULTS

A. PERFORMANCE

To test the performance of the system, 16384 8-bit integers were used to initialize the memory. The data points were chosen to be a half sine wave with amplitude of 100 to prevent negative values while at the same time scale the data so they can be represented by integers. By choosing a dataset that varies from point to point, a word transmitted out of turn cannot be seen as a correct transmission. For example, if a uniform data set was used, a lag of one data word is seen as a single transmit error, whereas in this case, all subsequent words are detected as out of turn.

When operating correctly, the system returns the data with no errors as plotted in Figure 6. The staircase in the plot comes from the 8-bit quantization error. The transfer timer is started when the read buffer is non-zero and ended when the expected number of data words is received. The amount of data transferred during this time is then the difference between the entire transfer size (16384) and the initial non-zero read buffer size. The transfer rate is the amount of data divided by the time elapsed.

The throughput for a RS-232 connection is set during initial connection. The average throughput is calculated by dividing the number of bits transferred by the amount of time of the transfer. The number of bits is set to 2^{14} data words multiplied by 16 bits per word, made up of 8 bits of data, 1 start bit, 1 stop bit, and 6 idle bits. The RS-232 waveform is plotted in Figure 7, with the expected 26 ms bit period.

The maximum transfer speed obtained via RS-232 with no errors was 36.9 kbps with a theoretical rate of 38.4 kbps (Table 1). By further increasing the data rate to 57.6 kbps, transmission errors were present (Figure 8). From the pattern of the errors, a loss of synchronization is likely to be the cause of the errors. The bits could be lagging in a manner that the 1's in the data are considered to be the terminating character. This would repeat until the data stream is lagged back to the corrected sequence. Note that the

additional 6 idle bits at the end of the sequence did not help to prevent errors in mid sequence. This will be discussed in Chapter IV.

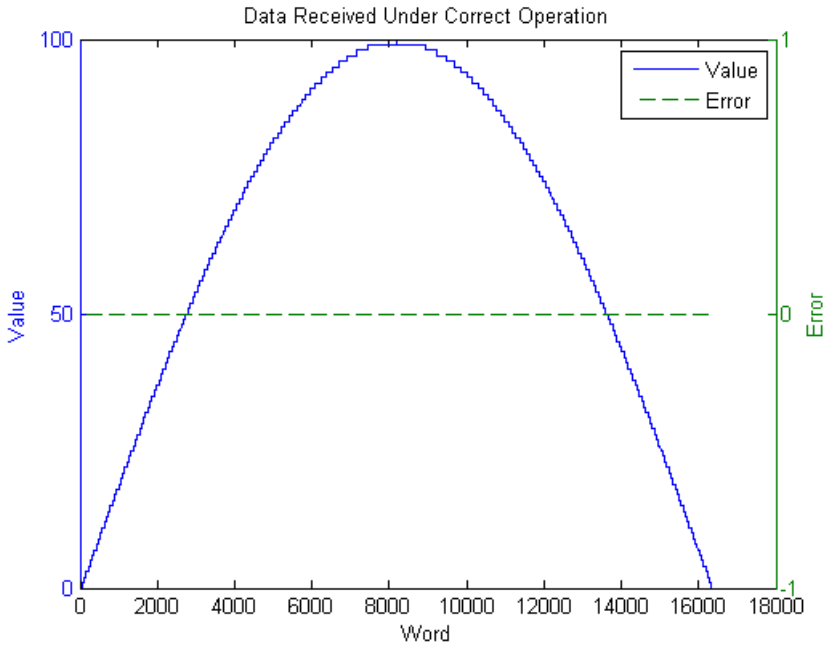


Figure 6. Data Received under correct operations. The error scale is on the right with 0 error for all data points. This was the case for RS-232 at 38.4 kbps, USB with 9-bit timing counter, and USB with TXE feedback.

Trial #	First nonzero buffer size	Buffer size before timer	Transfer Rate (kbits/sec)
1	17	17	36.835
2	67	67	36.862
3	23	23	36.855
Mean	--	--	36.851
Theoretical	--	--	38.400

Table 1. RS-232 transfer times at 38400 baud.

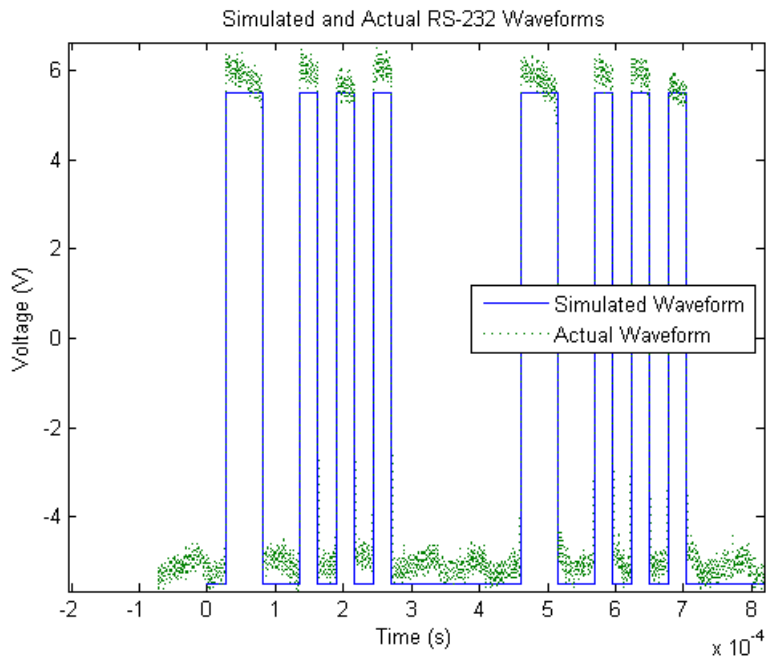


Figure 7. Simulated and actual RS-232 waveform at 38400 baud.

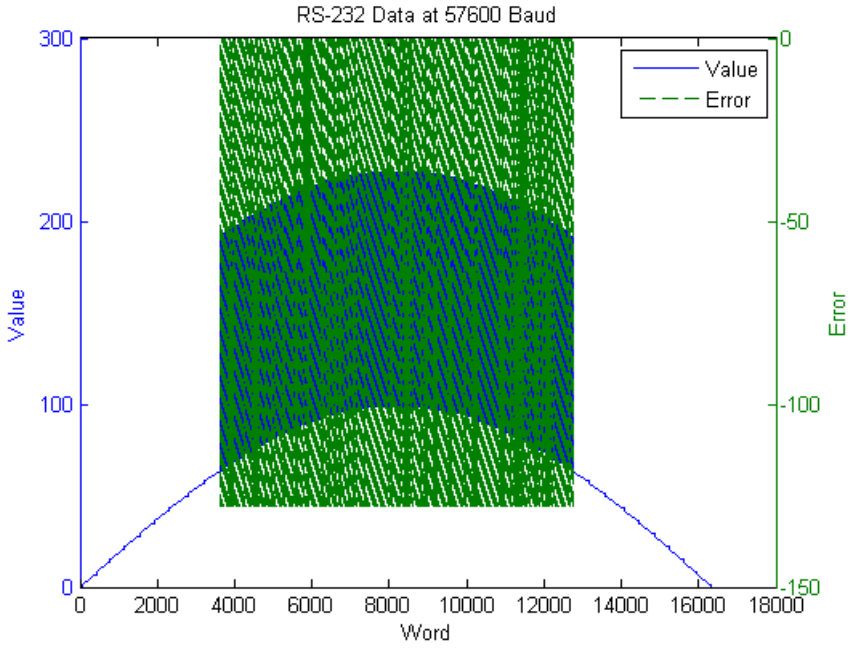


Figure 8. RS-232 data transfer at 57600 baud. Error scale is on the right.

The waveform of the fixed period USB transfer is plotted in Figure 9. The leading and trailing edges of the pulses are measured from the greatest overshot point and the results are summarized in Table 2. The pulse width of the WR pulse is 80 ns, which matches the design of 4 clock cycles with 20 ns per clock period. Similarly, the width of the BRAM counter is 20 ns, matching one clock cycle in the design. Lastly, the time between the when WR goes low and when the BRAM counter is enabled is 560 ns. This matches the designed delay between the falling edge of WR (4 clock cycle from the beginning of the timing counter) and the enabling of the BRAM address counter (at 32 clock cycles from the beginning of the timing counter), giving a delay of (32-4)*20 ns = 560 ns.

	Leading Edge	Trailing Edge
WR Pulse	17.93 us	18.01 us
BRAM Counter Enable	18.57 us	18.59 us

Table 2. Timing information for USB waveform.

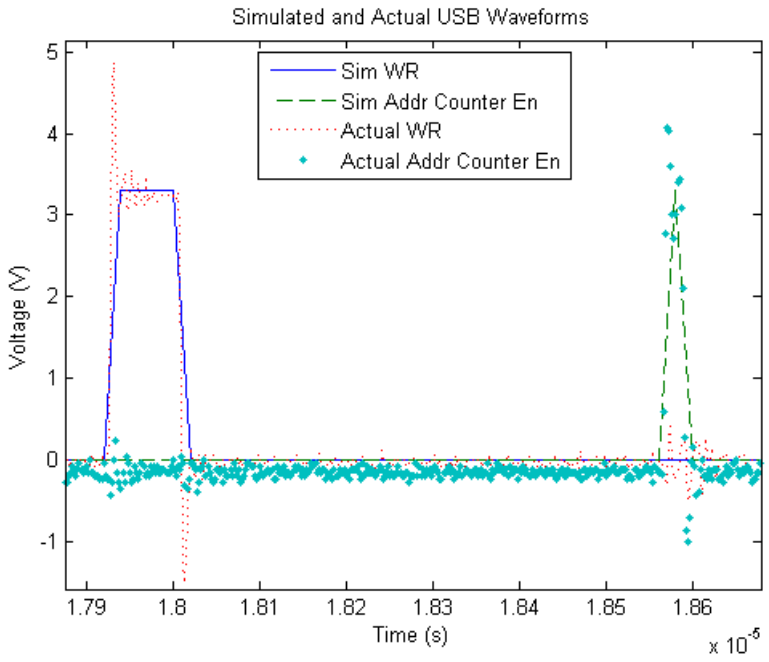


Figure 9. Simulated and actual waveforms for WR and BRAM address enable pins for fixed period USB transfer.

The theoretical USB transfer speed is calculated below for a 9-bit counter. Without the timing counter, 8 bits of data would be sent to the FIFO at each clock FPGA clock period of 20 ns. This would yield a theoretical transfer speed of:

$$\frac{\text{Data per clock cycle}}{\text{Clock period}} = \text{Max Data Rate} \tag{1}$$

$$\frac{8 \text{ bits}}{20 \text{ ns}} = 400 \text{ Mbps} \tag{2}$$

However, this is not attainable because the FIFO chip is only rated to transfer at a data rate of 8 Mbps. The timing counter is designed specifically to slow down the transfer rate caculated in equations (1) and (2) by enabling the WR pin only once every 2^n cycles, where n is the bit width of the counter. For the 9-bit timing counter, the data rate is theoretically

$$\frac{\text{Max data rate}}{\text{Counts per write cycle}} = \text{Data Rate} \tag{3}$$

$$\frac{400 \text{ Mbps}}{2^9} = 781.25 \text{ Kbps} \tag{4}$$

The start timer was initiated after data is detected in the read buffer, and terminated when the read buffer reaches the expected number of data points. The read command simply assigns the data in the queue, already on the PC, to a variable, so there is no need to include the read operation as a part of the transfer time.

Because the FIFO reads the data in blocks of 4k bytes, 4k words are already transferred to the PC by the time the timer starts [13]. This is shown in Table 3, where the data size before transfer is always the same, 3968 bytes. This makes sense because 2 control bytes are sent for every 62 data bytes, so a 4k bytes transfer would contain 3968 bytes of data. Because the fast transfer speed, additional data will flow between the time the first data packets are detected, and when the timer starts counting. However, due to the burst nature of the traffic, the queue will always fill up to a fixed size and pause while the next block is transferred. Therefore, the transfer rate is

$$\frac{\text{Data length - Initial Buffer Size}}{\text{Transfer time}} 8 \text{ bits per word} = \text{Transfer Rate} \tag{5}$$

Using equation (5) with 16384 as the data length, 3968 as the initial buffer size and the measured transfer times, the results in Table 3 were obtained. The theoretical value in the table is obtained from equation (3) and (4).

The performance of the chip operated below the expect speed in both cases. The average speed was 702.6 kbps, about 10% below the expected value. This is due to the last block of data not being 3968 words long. When the data first arrives in the PC, it is queued in the USB transfer buffer. When the USB transfer buffer reaches 3968 data words, it is appended to the read buffer. When the last 512 words are transferred to the USB transfer buffer, the PC waits for it to fill to 3968 words before appending it to the read buffer. Only after a timeout period, does the driver realize there are no more data to be transferred and appends the data to the read buffer. When the program is timed from the read buffer being 3968 words long to 15872 words long, the transfer speed increases to 781.9 kbps, slightly over the theoretical speed due to the resolution of the timer.

Errors in the transfer were present when the timing counter was reduced to 8 bits, as shown in Figure 10 and Table 4. Note that the position of the errors were around 4000 and 16000, the time when the driver was busy copying data from the transfer buffer to the read buffer.

Trial #	First non zero buffer size	Buffer size before timer start	Transfer Rate for 16384 words (kbits/sec)	Transfer Rate for 15872 words (kbits/sec)
1	2728	3968	704.385	780.012
2	814	3968	704.424	781.812
3	1219	3968	699.034	781.959
Mean	--	--	702.614 (781.93)	781.93
Theoretical	--	--	781.25	781.25

Table 3. USB transfer times using 9-bit timing counter.

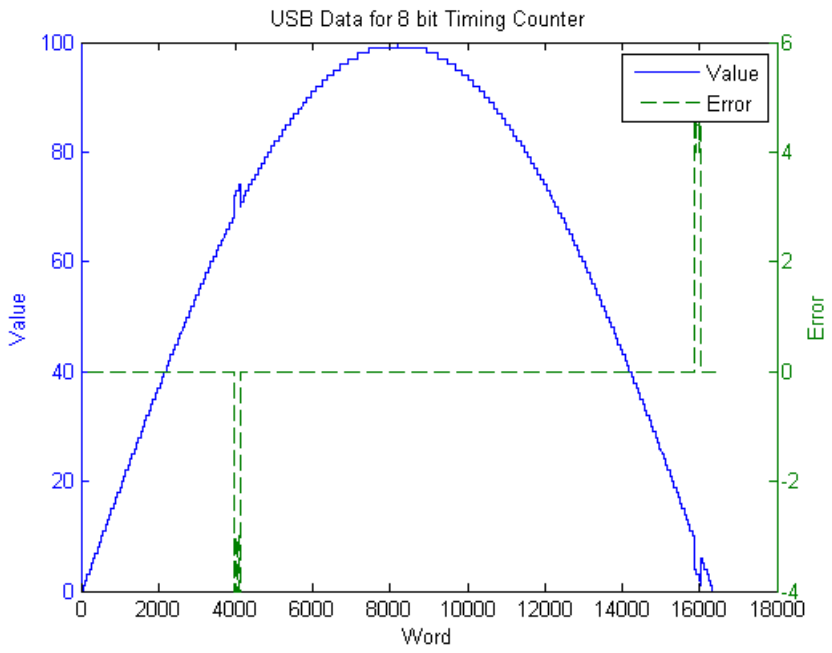


Figure 10. USB data transfer using 8-bit counter. Error scale is on the right with an average of 249 words in error.

Trial #	Transfer Time (sec)	Transfer Rate (kbits/sec)	Errors
1	0.125189	1047	288 (1.76%)
2	0.127032	1032	204 (1.25%)
3	0.126959	1032	256 (1.56%)
Mean	0.126393	1037	249 (1.52%)
Theoretical	0.087381	1500	0

Table 4. USB transfer times using 8-bit counter.

By using the TXE pin to regulate data flow, a transfer rate of 3.2 Mbps without error was achieved. The TXE and WR waveforms are plotted in Figure 11. The WR pulse is 60 ns in duration as expected, with 500 ns to 600 ns between pulses. As was the case above, when the data is packaged in three blocks of 3968 words, a transfer rate of 6.7 Mbps was reached (Table 5). The increase in data rate was more significant in this case because the waiting time for the last 512 words now make up for a significant

amount of total transfer time. The time it takes to transfer the last 512 words was obtained by subtracting the time it takes to transfer the three block 3968 words from the total time.

Total time = Time to transfer 1st 3x3968 words + time to transfer last 512 words (6)

$$\frac{(11904+512)*8}{\text{Overall transfer rate}} = \frac{11904*8}{\text{Transfer rate without last 512 words}} + t \qquad (7)$$

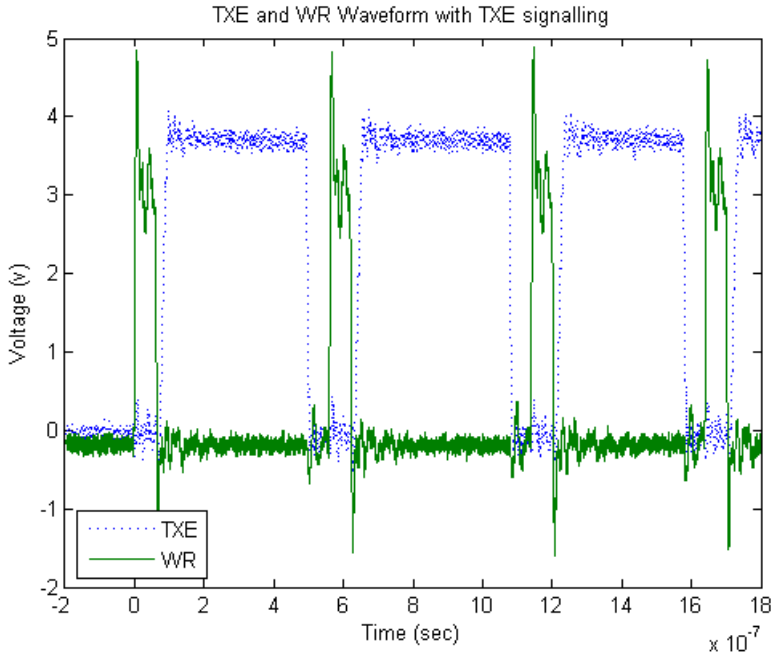


Figure 11. Signal using the TXE pin on the FIFO to control data flow.

Substituting 703 kbps for the overall rate and 782 kbps for the block transfer rate (Table 3) into equation (7), the waiting time for the last 512 words was calculated to be 0.020 seconds for the design with fixed write period. Similarly substituting 3.1 Mbps for the overall rate and 6.7 for the block transfer rate (Table 5), a t of 0.017 seconds was obtained for the design with TXE feedback. The time spent waiting for additional data before timing out was consistent in both designs.

B. RESOURCES USED

More resources are used on the FPGA for the USB design because additional logic is required to trigger the WR bit on the FIFO at the correct time (Table 6). The RS-232 design is simpler because there is no such data flow control required and control bits are simply passed along the output sequentially with the data bits. The complexity of the FIFO itself does not lead to more resources used directly. Should it be completely integrated on the FPGA, the design would still be similar. This is evident in that the onboard RS-232 interface does not reduce the complexity of the RS-232 design. Should there be an external RS-232 chip, the output gateway would simply change pin assignments and the complexity of the RS-232 design would remain the same. Note that the difference in resources taken in the various designs are insignificant compared to the performance difference. Although the design is more complex for the USB FGPA models, the performance and reliability of the USB design out weights such complexities. Although the USB with TXE feedback model appears more complex, it was implemented with fewer resources on the FPGA. Again, the cost resources taken for a better design was insignificant compared to the performance increase.

Trial	First non zero buffer size	Buffer size before timer start	Transfer Rate for 16384 words (kbits/sec)	Transfer Rate for 15872 words (kbits/sec)
1	3968	3968	3147.454	6795.345
2	3968	3968	3162.001	6805.266
3	3968	3968	3195.331	6636.264
Mean	--	--	3168.262	6745.625
Theoretical	--	--	8000.000	8000.000

Table 5. USB Transfer times using TXE pin.

	RS-232	USB (Fix WR period)	USB (TXE Feedback)
# of flip flops	45	55	47
# of LUTs	18	27	26
# of slices	35	42	37

Table 6. Resources used.

C. CHAPTER SUMMARY

This chapter presents the results obtain from the three designs. The next chapter will analyze the significance of these results.

IV. ANALYSIS

The USB solution outperforms the RS-232 solution as expected with slightly more logical resources used. The fastest possible speed between the FPGA and the PC with the USB protocol is specified by the manufacture at 8 Mbps, whereas the fastest possible RS-232 supported by Matlab is 256 kbps, about 32 times slower. In this thesis, 3.17 Mbps USB transfer and 36.9 kbps RS-232 transfer were achieved. Both speeds exceeded the 32 kbps requirement for a dual 16-bit, 10k data point transfer every 10 seconds in the original specifications. However, to take into account the time that the data need to be written into memory and other overhead, the RS-232 solution could be impractical whereas the USB solution remains viable.

The RS-232 matches its theoretical performance within (38400-36851) / 38400 = 4.0%. The RS-232 method requires reconfiguring every time the baud rate changes. The rate of transfer is controlled by the frequency of timer, which needed to be specified at compile time. The value of the frequency was also determined empirically. For the 38400 bps, the theoretical scaling factor should be 50 MHz / 38400 = 1295. However, the scaling factor was determined empirically be 1355 by measuring the bit period on the oscilloscope. This discrepancy between the theoretic scaling factor and its actual value leads to more time spent designing the circuit, compared to the USB design where the rates are auto negotiated based on the frequency of the control counter and does not need to be calculated before compilation.

The 6 idle bits in the RS-232 model adds 6/16 = 37.5% overhead to the communications channel. It is interesting to note that as shown in the 57.6k transfer case, the extra padding at the end does not provide extra syncing functionality. The data in the middle of the set contained transfer errors despite the fact that the extra idle time in between words intended to reset the line. This is due to the fact that the extra padding bits are seen as data once the initial bit is lagged. For example if the 1 in brackets in the string 1111_0101010[1]01_111 is interpreted as two 1's, then the extra idle signals will

simply be interpreted as a part of the next data word. Therefore, syncing the line after every bit was not effective as shown, especially with 37.5% performance penalty.

For the USB case, error occurs when an eight bit counter was used to try to achieve the transfer speed to 1.56 Mbps. This is because the FIFO was not ready to receive data while the driver is copying data to the read buffer. This can be seen in Figure 10, where the error occurs at multiple of 4000. At slower data rates, errors are not present because the driver can carry out the copy operation before the FPGA issues another WR enable. For the 9-bit counter, the successive WR pulses comes in periods of every $20*2^9$= 10.24 us, where as for an 8-bit counter, the WR pulses comes in periods of every $20*2^8$= 5.12 us. Therefore, the USB driver needs a minimum of 5.12 us to write the data, but not more than 10.24 us. However, the errors due to the driver being busy does not propagated after the driver is free. This is because the WR pulse and the data are sent separately. Even if a stray WR signal is sent when the data was not ready to be read, once the FIFO is ready again, the previous errors does not impact the current transmission. This is shown in Figure 10, where once the buffer is copied, the data continue to transfer correctly. This is a main advantage over the RS-232 design, where one error bit will continually propagate and causes subsequent errors

There is no error encountered when using the TXE pin to regulate the transfer, and the transfer rate is the maximum possible. The overall data rate is highly dependent on whether the data size is a multiple of 3968. With 0.2 seconds of time out length, it will be faster to pad the data to 3968 blocks and use an extra 0.066 seconds to transfer the whole block when the TXE pin is used. When the TXE pin is not used, padding the results will result in an additional 0.57 seconds of transfer time, exceeding the 0.2 seconds of timeout. Therefore, not only is using the TXE pin result in an average of 4.5 times faster performance, but have the potential to be faster by 8.6 times with the correct data structure.

With TXE feedback, the FPGA transfers the data at approximately 8 bits every 600 ns (Figure 11), with a theoretical data rate of 13.3 Mbps. Although slowdowns could occur in other parts of the transfer, the majority of the slow down comes from the PC processing the buffer in software, which restrict the transfer rate to the 8 Mbps specified

by the manufacture. To further speed up the transfer, the overhead in the Matlab library function calls can be eliminated by using C. However, the complexity of doing so does not justify the last bit of performance increase for current applications.

For the resources used, The main source of the increased logic comes from the fact that the USB design have to trigger WR signals to the FIFO in addition to handling the data transfer. In the RS-232 case, the control signals are incorporated into the data as start and stop bits. Once the correct rate is determined, the control signals are treated just as ordinary data. In the USB case, the data and control signals reside on separate physical pins, so the timing between the two have to be coordinated within the logic. This is readily apparent when the RS-232 design could be implemented using one counter where the USB design required two counters to handle the control signals correctly.

This chapter explained why errors are present in the transfer rates. The next chapter will present conclusions and future work.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS

A. GENERAL COMMENTS

Using RS-232 would not be a viable solution for high speed data transfers. For example, to off load data from a digital sensor, the transfer rate of 32 kbps was barely accomplished with an actual transfer rate of 36.9 kbps. Although it could still be feasible by cropping off the extra syncing bit, the USB solution has already been demonstrated to be a viable option at 3.17 Mbps. In addition, the speed using the USB method was achieved without the need to match the expected transfer rate on the PC side with the bit period on the FPGA. This also eliminated the need to measure the period of the waveform on an oscilloscope to empirically determine the period of the counter needed to control the RS-232 data transfer. In the end, the solution developed here is easy to use and maintain, without risking proprietary vendor lock.

B. ALTERNATIVE SOLUTIONS

Alternatives also exist from the solution built in this thesis. Simulink with System Generator and ISE were used to program the FPGA to send the data. On the receiver end, Matlab was used to program the PC to read the data. This choice is made for ease of maintenance. A software peripheral core could also be built with XPS which would automate the output from memory to the PC via C style print commands [17]. This would eliminate the timing requirements in the design but the end product will be limited to RS-232 communications only, since the program can only output to UARTs such as the serial port. Also, the FPGA could be programmed from scratch in a HDL such as VHDL or Verilog, but would require substantially more time to code, debug, and maintain.

More choices are available for the program on the PC to read the data, programming on the PC is more mature than programming on the FPGA. There are various projects posted by the FTDI to work with the FIFO, including popular languages such as the various forms of C, LabView, Python, and Perl [9]. One disadvantage of

these is that a software development environment would need to be installed explicitly for this purpose with no real added benefit. One exception is code written in C and compiled with a light compiler. This removes the need for Matlab for reading the data and replaces it with a much smaller C compiler. This is demonstrated to be successful under Linux. However, due to the way the operating system handles integers, a valid USB handle cannot be obtained under Windows. Also, due to currently available platforms, dedicating one Linux machine for just data acquisition is not logistically viable.

Also, complete bundled solutions exist [6], [7], [8]. These had the benefit of working out of the box with commercially available support. However, direct licensing costs and restrictions on future development makes it unattractive for academic development work. However, there are also solutions which provide a USB connector directly on the FPGA and the necessary header files and libraries without the restrictive licensing that could be used.

Lastly, different protocols could be used to transfer the data, such as Ethernet [16]. Currently, 100Mbps Ethernet connections are available on FPGAs which could increase the throughput. However, this would require the data to be sent over the LAN, which could be slower depending on network activity. Gigabit Ethernet could also greatly boost transfer speed. However, with the advent of USB 3.0, which has a throughput of 5 Gbps, USB once again will be the most optimal for data transfer [18].

C. FUTURE WORK

For the RS-232 model, performance could be increased by discarding the 6 idle bits. This would increase performance with no additional errors. To provide better synchronization, either RTS/CTS hardware handshakes or XON/XOFF software signals could be used to control the flow of data. This is similar to using the TXE pin in the USB case.

For the dataset themselves, mechanisms can be setup to accept 16-bit negative float point numbers. Depending on the data source, multiple encoding schemes could be used to convert real numbers into integers.

For better functionality, a software and external trigger can be developed to remove the operator from the vicinity of the board. This would eliminate the operator having to come in contact with a high voltage power supply and also pave the way to sending a variable word length as another parameter.

D. ADDITIONAL APPLICATIONS

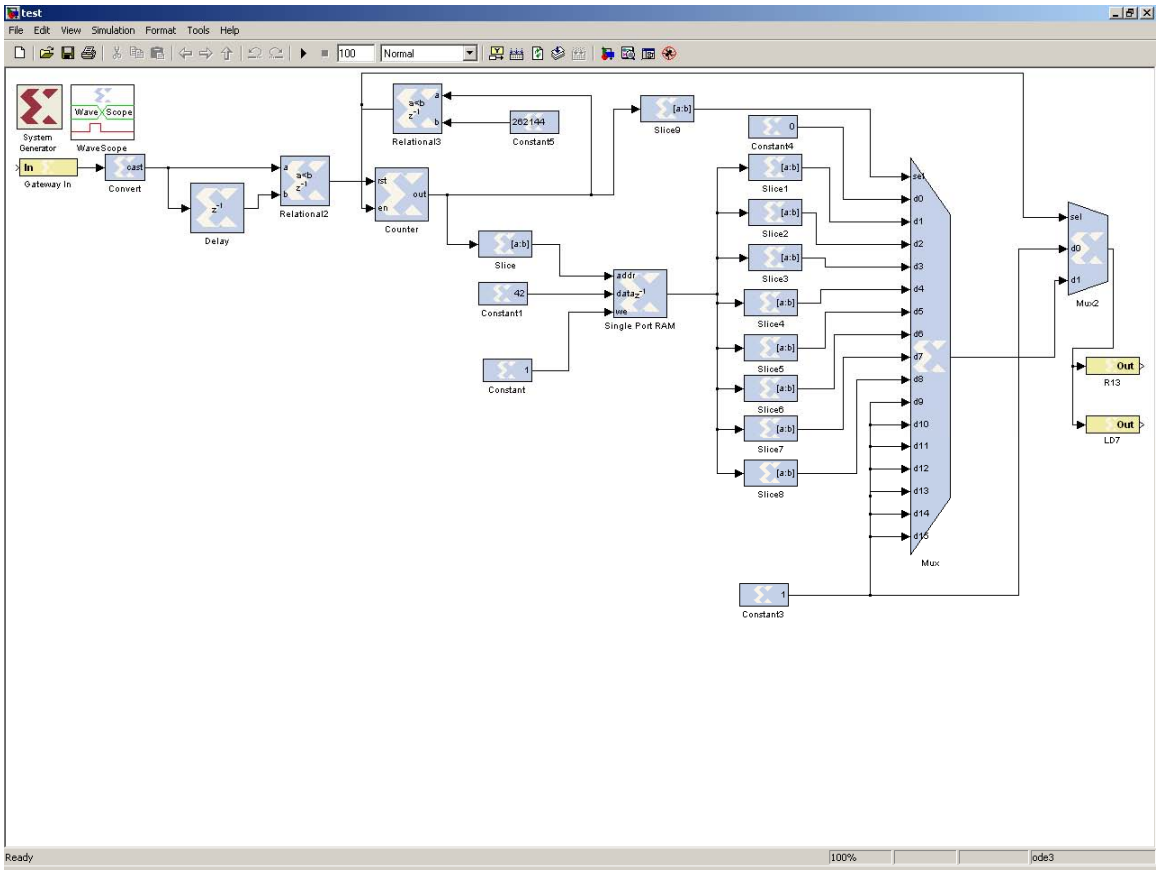
The original motivation for this thesis was to offload test data from digital sensors to a PC. However, the applications developed in this thesis are generally applicable. Future projects will only need to write data to the BRAM and provide a trigger signal for the data to be stored on the PC and analyzed with a much wider set of tools, such as Matlab, than available on the FPGA.

Possibilities also exist for bidirectional communications where the PC can send signals to the FPGA to control another set of hardware. Especially with the ease of RS-232 programming, low data rate applications such as remote control for digital sensors, automated homes, and electric ships can be developed with the same ground work. For high data rates, this thesis could be used for testing data storage applications.

This thesis provides the low level interface to the memory available on an FPGA. Higher level protocols and applications can use this to manipulate data with the variety of functions available on the PC via Matlab. Graphical user interfaces could also be refined to be more user friendly. Since the model was developed in Simulink, it is portable across multiple FGPA vendors with modifications to the pin assignment, as was done when porting from the Spartan development board to the Opal Kelly board. With the lower level communications accomplished, higher level applications can be built on top to support a wide array of applications.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. RS-232 DESIGN



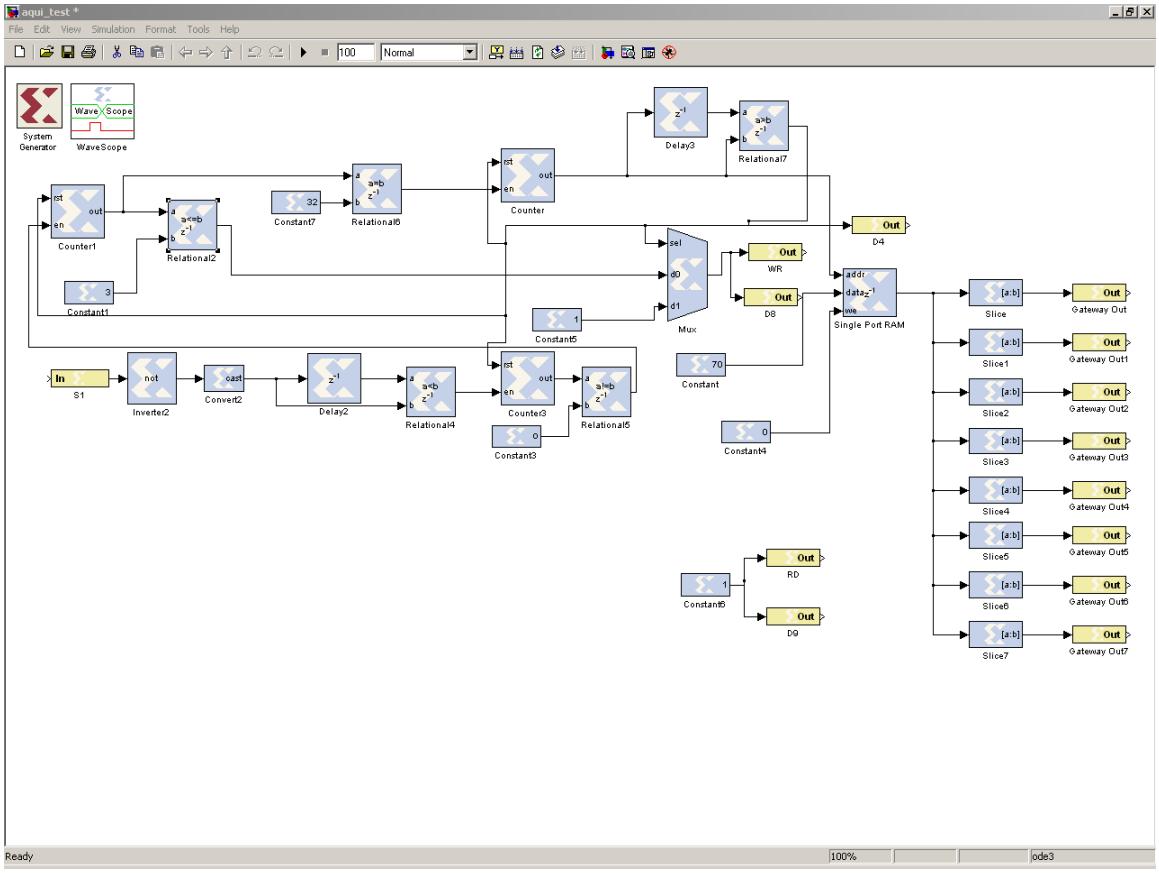
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. MATLAB PROGRAM FOR RS-232

```
lim=2^14;
s=serial('COM3', 'BaudRate', 38400);
s.InputBufferSize=lim;
s.Timeout=60;
fopen(s);

buff=0;
while (buff == 0)
    buff=s.BytesAvailable;
end
buff
buff1=s.BytesAvailable
tic;
while (s.BytesAvailable < lim)
end
dt=toc;

m=fread(s, lim);
xfer_rate=(lim-buff)*16/dt

pause(5);
if (s.BytesAvailable > 0)
    y=fread(s, s.BytesAvailable);
end

fclose(s);
```

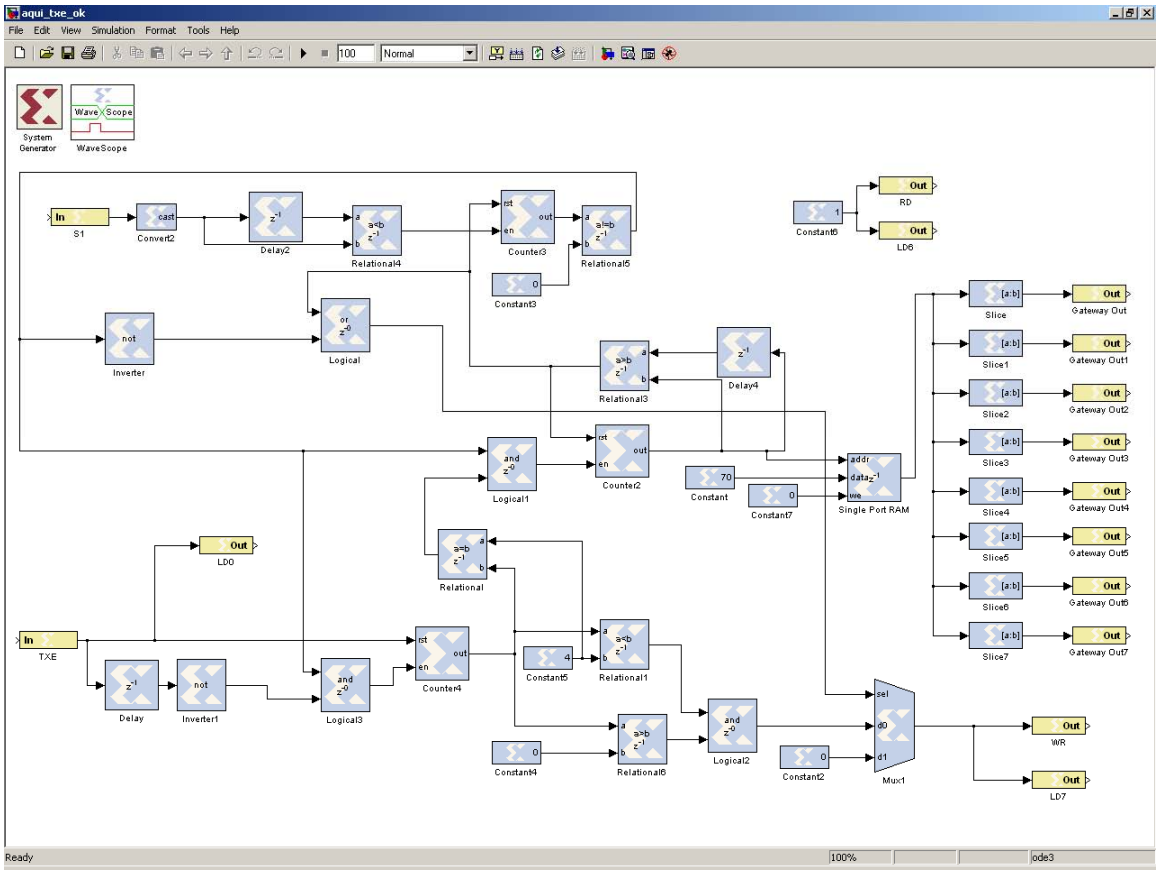
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. USB DESIGN USING A FIXED WRITE PERIOD



THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. USB DESIGN USING TXE FEEDBACK



THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. MATLAB PROGRAM FOR USB

```
buff_size=2^14;

loadlibrary('lib/ftd2xx.dll', 'include/ftd2xxM.h');

h=libpointer('uint32Ptr', uint32(0));

disp('opening');
s=calllib('ftd2xx', 'FT_Open', 0, h);
if (s ~= 0)
    disp('open error');
    return;
end

disp('set baud rate');
s=calllib('ftd2xx', 'FT_SetBaudRate', h, 9600);
if (s ~= 0)
    disp('set baud rate error');
    return;
end

s=calllib('ftd2xx', 'FT_Purge', h, 1);

dwRxSize=0;
pdwRxSize=libpointer('uint32Ptr', dwRxSize);

v=0; u=0; w=0; r=0; t=0;
pu=libpointer('uint32Ptr', u);
pw=libpointer('uint32Ptr', w);
pr=libpointer('uint32Ptr', r);
pt=libpointer('uint32Ptr', t);


dwBytesRead=0;
pdwBytesRead=libpointer('uint32Ptr', dwBytesRead);

disp('get queue status');
disp('press button');
buff=0;
sz=[];
tic;
while (buff == 0)
    s=calllib('ftd2xx', 'FT_GetQueueStatus', h, pdwRxSize);
    buff=pdwRxSize.Value;
    sz(end+1)=buff;
end

buff
tic;
s=calllib('ftd2xx', 'FT_GetQueueStatus', h, pdwRxSize);
```

```
buff1=pdwRxSize.Value

while ((pdwRxSize.Value < buff_size) && s == 0)
    s=calllib('ftd2xx', 'FT_GetQueueStatus', h, pdwRxSize);
end
dt=toc;
xfer_rate=(buff_size-buff1)*8/dt

pcBufRead=75*ones(1, pdwRxSize.Value/4);
ppcBufRead=libpointer('uint32Ptr', pcBufRead);

data=70*ones(1, pdwRxSize.Value/4);
pdata=libpointer('uint32Ptr', data);

% [v, pu, pw, pr, pt]=calllib('ftd2xx', 'FT_GetStatus', h, pr, pt, pu)
% [u, pr, pdata, pt]=calllib('ftd2xx', 'FT_Read', h, ppcBufRead, pw,
pdwBytesRead);

disp('reading');
[u, pr, pdata, pt]=calllib('ftd2xx', 'FT_Read', h, ppcBufRead,
pdwRxSize.Value, pdwBytesRead);
if (s~=0)
    disp('read error');
    return;
end

z=typecast(uint32(pdata), 'int8');
disp('closing');
s=calllib('ftd2xx', 'FT_Close', h);

unloadlibrary('ftd2xx');
```

APPENDIX F. MODIFIED FTD2XX.H

Added:	#include "WinTypes.h"	
Changed:	typedef PVOID FT_HANDLE; typedef DWORD *FT_HANDLE;	to
Change:	LPVOID lpBuffer, LPDWORD lpBuffer, FT_Read	to in function

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] M. Tooley, *Newnes Data Communications Pocket Book*. Oxford: Newnes, 1997.

[2] J. Axelson, *USB Complete*. Madison, WI: Lakeview Research, 2005.

[3] SRC Corporation, "SRC Tutorial Exercise," 2005.

[4] J.H. Park, et al., "Implementation of a Stand-alone Color Image Transfer Circuit Using the USB 2.0 Interface," *International Conference on Control, Automation and Sytems*, 2007.

[5] Project, "FGPA USB." Retrieved Nov 3, 2009 from www.fpgaz.com/usbp.

[6] P. Warnes, *Hunt Engineering RTG001. Ready to Go System for Data Acquisition and Processing Users's Manual*. 2006. Retrieved Nov 3, 2009, from http://www.hunt-rtg.com/pdf/rtg001.pdf.

[7] Orange Tree, *ZestSC1: FPGA + USB Module*. Retrieved Nov 3, 2009, from http://www.orangetreetech.com/pdf/ZestSC1Flyer.pdf

[8] O. Kelly, *XEM 3010 User's Manual*. 2009. Retrieved Nov 3, 2009, from http://www.opalkelly.com/library/XEM3010-UM.pdf

[9] FTDI. "Code Examples." Retrieved Nov 3, 2009 from http://www.ftdichip.com/Projects/CodeExamples.htm.

[10] Xilinx, *Spartan-3 FPGA Starter Kit Board User Guide*, user manual, 2008.

[11] FTDI, *UM245R USB-Parallel FIFO Development Module*, user manual, 2005.

[12] D. Olney, "USB + FTDI + Compiler." Matlab Central. May 2005. Retrieved Nov 3, 2009, from http://www.mathworks.de/matlabcentral/newsreader/view_thread/86843

[13] FTDI, "Optimizing D2XX Data Throughput," 2006.

[14] Proffitt, J, et al., "A Flexible High-Rate USB2 Data acquisiontion Sytem for PET and SPECT Imaging." *IEEE Nuclear Science Symposium Conference Record*, 2005.

[15] W. Zhou, "Design of Serial/Parallel Communication Interface of Digital Potentiometer with PC." *The Eighth International Conference on Electronic Measurement and Instruments*, 2007.

[16] G. Singh and J. Conrad, "Easy-to-Use Communication Interfaces for Data Acquisition." *IEEE Southeastcon*, 2008.

[17] Xilinx. *EDK Concepts, Tools, and Techniques*, user manual, 2009.

[18] USB Implementer's Forum. *Universal Serial Bus 3.0 Specification*, user manual, 2008.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Jeffrey B. Knorr
   Department of Electrical and Computer Engineering
   Monterey, California

4. Professor Alexander Julian
   Department of Electrical and Computer Engineering
   Monterey, Calinfornia